INSTITUTO FEDERAL GOIANO – CAMPUS MORRINHOS

CURSO SUPERIOR DE TECNOLOGIA EM SISTEMAS PARA INTERNET

MARCELO RODRIGUES SILVA

DA CRIAÇÃO A ANALISE DE FRAMEWORK DE ORM.

MORRINHOS – GO

2016

MARCELO RODRIGUES SILVA

DA CRIAÇÃO A ANALISE DE FRAMEWORK DE ORM.

Monografia apresentada ao Curso superior de Tecnologia de Sistemas para Internet do Instituto Federal Goiano – Campus Morrinhos, como requisito parcial para obtenção de título de Tecnólogo em Sistemas para Internet.

Orientador: Msc. Marcel da Silva Melo.

MORRINHOS – GO

2016

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema Integrado de Bibliotecas – SIBI/IF Goiano Campus Morrinhos

S586d Silva, Marcelo Rodrigues.

Da criação a análise de Framework de ORM. / Marcelo Rodrigues Silva. – Morrinhos, GO: IF Goiano, 2016.
58 f. : il. color.
Orientador: Me. Marcel da Silva Melo.

Trabalho de conclusão de curso (graduação) – Instituto Federal Goiano Campus Morrinhos, Tecnologia em Sistemas para Internet, 2016.
1. Framework. 2. Desenvolvimento de software. 3. Repositório de código. I. Melo, Marcel da Silva. II. Instituto Federal Goiano. Tecnologia em Sistemas para Internet. III. Título

CDU 004.4

MARCELO RODRIGUES SILVA

DA CRIAÇÃO A ANALISE DE FRAMEWORK DE ORM.

Data da defesa: 21/12/2016

Resultado: _____

BANCA EXAMINADORA ASSINATURAS

Marcel da Silva Melo Profº Msc. _____
Instituto Federal Goiano Campus Morrinhos

Jose Pereira Alves Profº Esp. _____
Instituto Federal Goiano Campus Morrinhos

Norton Coelho Guimarães Profº Msc. _____
Instituto Federal Goiano Campus Morrinhos

MORRINHOS – GO

2016

DEDICATÓRIA

Dedico este trabalho a todas as pessoas que sempre estiveram do meu lado, não só nos momentos felizes, mas nos momentos de dificuldades também.

Agradecimentos

Quero agradecer, em primeiro lugar, a Deus, pela força e coragem durante toda esta longa caminhada.

À minha família pela capacidade de acreditarem em min, pelo amor, incentivo e apoio incondicional.

Ao meu orientado Msc. Marcel Melo da Silva, pela paciência na orientação e incentivo que tornaram possível a conclusão desta monografia.

RESUMO

Devido ao grande aumento na demanda de desenvolvimento de software é necessário utilizar métodos para agilizar o processo de desenvolvimento de software, e um dos métodos mais conhecidos é utilização de *frameworks*. O objetivo desse trabalho é criar um *framework* que automatize o trabalho do desenvolvedor, essa automatização é realizada pela utilização de repositório de códigos que facilitaram o desenvolvedor em tarefas rotineiras, permitindo ao desenvolvedor focar nas regras de negócio do sistema. Além disso, o *framework* tem função de facilitar o trabalho do desenvolvedor que trabalhe com arquiteturas de "n" camadas ou arquitetura de microserviços.

Palavras-chave: *Framework;* Desenvolvimento de *Software*. Repositório de código

ABSTRACT

Under the grat growth in the demand of software developer it's necessary use some methods to facilitate software development process, and one of the most known methods is the usability of framework. the objective of this work was creation of a framework automates developers work, this can happen using code repository and will facilitate the developers in your routine tasks, allowing to focus in system business rules. In addtion the framework have the function to facilitate the developer work which work with architectures with “n” sections or architectures of micro-services.

Keywords: Framework, Software Development, Code Repository.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

1.1 Contextualização

O conceito de *framework* na programação surgiu quase que juntamente com a programação orientada a objeto. Em 1967, uma linguagem de programação chamada *SIMULA* (DAHL; NYGAARD, 1966) foi criada com objetivo de criar software para simulação. A *SIMULA* não foi muito disseminada mas trouxe diversos conceitos que se utiliza na programação moderna, como: objetos como classes, objetos, encapsulamento de dados, herança, métodos e associações dinâmicas (CARNEIRO, 2003, p.4).

A *SIMULA* tinha permanecido desconhecida até os anos 70. Nessa década, a companhia Xerox Parc redescobriu as suas ideias e iniciou a criação de nova linguagem de programação chamada *Smalltalk* (ABDALA,2012). Por ter incorporado a linguagem *SIMULA*, havia um conjunto de classes cooperantes que podem ser descritas como *framework* (CARNEIRO, 2003, p.4).

De acordo Carneiro (2003, p.4) "O primeiro *framework* largamente utilizado foi o *framework* de interface do usuário (UI) do Smalltalk-80 chamado de *Model-View-Controller*.".

1.2 Objetivos

O objetivo desse trabalho é demostrar a elaboração de um *framework* de ORM (Object-relational mapping) utilizando a linguagem de programação C# com a IDE (*Integrated Development Environment*) de desenvolvimento Microsoft Visual Studio. Será apresentado as principais características de um *framework* e seu fluxo de funcionamento. Para demostrar seu fluxo de funcionamento será apresentado um protótipo no qual será utilizado as principais funcionalidades do *framework* e o mesmo será comparado com um *framework* do mercado.

Os principais objetivos esperados para o *framework* é a geração automática de SQLs simples que funcionem perfeitamente em diferentes SGDBs (Sistema de Gerenciamento de Banco de dados) mantendo o padrão de cada um, os SGDBs suportados pelo *framework* são: Miscrosoft SqlServer (MAGALHÃES,2015), Oracle (WATSON,2009), MySql (MILANI,2007), PostgreSQL (MILANI, 2008). Além disso, o *framework* deve facilitar o

trabalho do programador minimizando o tempo gasto com funções básicas do sistema, e o uso de várias conexões com banco de dados diferentes.

1.3 Motivação

A motivação para esse trabalho foi a automatização do desenvolvimento de sistema. O *framework* baseia-se na criação de repositórios de códigos, repositórios de código e padrão de projeto ao qual possui classes com métodos genéricos que podem ser reaproveitados em outras classes ou projetos. Tais repositórios devem facilitar tarefas rotineiras do desenvolvedor, como comunicação com banco de dados, permitindo assim que o mesmo foque na criação da regra de negócio exclusivas de cada sistema.

Espera-se criar um *framework* que realmente facilite o trabalho para o desenvolvedor. Com um funcionamento simples, o desenvolvedor deverá apenas herda as classes do *framework* e mapear as entidades de acordo com o SGDB. O *framework* terá a responsabilidade de tratar as demandas necessárias para que funções básicas sejam geradas dinamicamente, permitindo ao desenvolvedor focar na regra de negócio do sistema.

Além disso, busca-se a criação de um *framework* que facilite para o desenvolvedor trabalhar com vários bancos de dados em um mesmo projeto, como acontece em arquitetura microserviços (WOLFF,2016). Após o mapeamento das classes, o desenvolvedor não precisará se preocupar com o gerenciamento desses vários SGDB, pois essa tarefa será de responsabilidade do *framework.*

1.4 Estrutura do trabalho

O trabalho está organizado em cinco capítulos, incluindo o presente capítulo de introdução. O capítulo 2, apresenta os conceitos e as ferramentas utilizadas para desenvolvimento do *framework*. O capítulo 3, descreve as principais funcionalidades do *framework*. O capítulo 4, descreve a atualização do *framework* e apresenta um comparativo realizado entre o *framework* proposto neste trabalho e um *framework* do mercado. O capítulo 5, apresenta as considerações finais e as conclusões obtidas no desenvolvimento do trabalho.

2 REFERENCIAL TEÓRICO

Para uma melhor compreensão da criação de um *framework* é necessário o conhecimento de algumas definições importantes sobre o conceito de *framework* e as ferramentas necessárias para criação do mesmo.

2.1 Framework

O *framework* é um conjunto de objetos reutilizáveis que engloba conhecimentos de determinadas áreas e se aplica a um domínio especifico (CARNEIRO, 2003, p. 5). Segundo (JONHSON e FOOTE, 1998) um framework é:

> "Definido também como um projeto de reutilização que descreve como o sistema está decomposto em um conjunto de objetos que interagem entre si. O sistema pode ser uma aplicação inteira ou apenas um subsistema. O framework descreve a interface de cada objeto e o fluxo de controle entre eles. "

Os *frameworks* podem ser utilizados em diversas áreas, no caso desse trabalho será desenvolvido um framework para ênfase no desenvolvimento de software.

> "No ambiente da programação orientada a objetos, os frameworks são compostos por interfaces e classes abstratas e a sua instanciação, uso, ocorre através da especialização ou composição dos seus serviços. Por isso, pode-se afirmar que os frameworks são compostos por pontos fixos – frozen spots - ou também conhecidos como hook points, que são serviços já implementados pelo framework que normalmente realizam chamadas indiretas aos pontos host spots, que são funcionalidades, serviços, e que devem ser implementados, através da característica da realização – herança – pelos desenvolvedores que irão inserir os seus códigos inerentes ao domínio da aplicação." (LEITE, 2006)

Segundo (FAYAD, 1999), citado por (LEITE, 2006), a utilização de *frameworks* apresenta as seguintes vantagens:

I. Melhora a modularização – encapsulamento dos detalhes voláteis de implementação através de interfaces estáveis;

II. Aumenta a reutilização – definição de componentes genéricos que podem ser replicados para criar novos sistemas;
III. Extensibilidade – favorecida pelo uso de métodos *hooks* que permitem que as aplicações estendam interfaces estáveis;
IV. Inversão de controle – IoC – o código do desenvolvedor é chamado pelo código do *framework*. Dessa forma, o *framework* controla a estrutura e o fluxo de execução dos programas.

Além dos conceitos apresentados anteriormente, a linguagem de programação C# foi escolhida para criação do *framework* proposto nesse trabalho. Os conceitos relacionados a linguagem de programação C# serão apresentados na próxima seção.

2.2 C#

O C# é uma linguagem de programação criada para desenvolvimento de aplicações que são executadas pelo .*NET Framework.* O C# é simples, poderoso, fortemente tipado e orientado a objeto (MICROSOFT, 2016).

A linguagem de programação C# foi desenvolvida pela Microsoft junto com a plataforma .NET no ano de 1999 por uma equipe de desenvolvimento formada por Anders Hejlsberd, que foi um dos criadores do Turbo Pascal (SURHONE, 2010) e o Delphi (JORGE,2013). Seu nome originalmente seria *cool*. Contudo, quando o projeto .NET foi apresentado ao público, o nome passou a ser C# (PORTAL EDUCAÇÃO, 2008).

A sintaxe do C# foi baseada no C++ e outras linguagem como Object Pascal (AVILLANO, 2009) e Java (COELHO, 2014). Ela se tornou o símbolo .NET por ter sido desenvolvida praticamente do zero, justamente, para funcionar na nova plataforma .NET, sem a preocupação de compatibilidade com código legados (PORTAL EDUCAÇÃO, 2008).

De acordo com (SANT'ANNA, 2000) no C#, "os objetos e os *arrays* são necessariamente alocados dinamicamente no *heap* com o uso do operador *new*. O índice dos *arrays* começas com zero e sua faixa e sempre verificada em tempo de execução".

"O C# inicializa a maioria das variáveis com zero e efetua diversas verificações de lógica, como se uma variável foi atribuída antes de ser usada se um parâmetro de saída foi atribuído e se um inteiro teve sua faixa violada" (SANT'ANNA, 2000).

“No C# existem outros tipos de *loop* além dos oriundos do C (*for*,*while*, *do while*). O *foreach* usado para varrer todos elementos de um *array* ou coleção. O único mecanismo de tratamento de erro do C# é a *exception”* (SANT’ANNA, 2000).

O C# suporta sobrecarga de métodos e operadores como no C++, contudo não existe o argumento *default*. Ele possui operadores de conversão, entretanto existe uma sintaxe para saber se uma conversão implícita ou explícita (SANT’ANNA, 2000).

Todos os objetos no C# têm um ancestral comum, o System.Object. No C# só é possível herança simples, ou seja, herdar apenas de uma classe. Contudo, é possível implementar várias interfaces, trazendo vantagens da herança múltipla sem muitos de seus problemas. Uma interface funciona como se fosse uma "classe abstrata", que possui apenas protótipos de métodos, sem nenhuma implementação (SANT’ANNA, 2000).

“No C# pode-se declarar *properties*, que funcionam sintaticamente como campos, mas na verdade chamam um par de métodos para atribuir ou receber o valor da *property*” (SANT’ANNA, 2000).

Os métodos por padrão não podem ser sobrescritos. Para sobrescrever um método no C# é necessário utilizar a palavra reservada virtual. Existe um protocolo específico para indicar se um método é derivado e reimplementa um método virtual (*override*) ou torna não virtual (*new*). No C# pode-se atribuir atributos às classes e métodos. Os atributos funcionam como uma diretiva de compilação, mas são resolvidos em tempo de execução (SANT’ANNA, 2000).

No C# é possível criar um objeto de uma classe dado apenas seu nome, atualizar propriedades utilizando seu nome e valor e chamar métodos utilizando seu nome e argumentos. Isso tudo é possível devido ao tratamento de informações em tempo de execução utilizando *reflections* (SANT’ANNA, 2000).

O C# possibilita criar ponteiros para métodos utilizando a palavra-chave *delegate*. Um *delegate* contém o endereço da função e método que a implementa. Os *delegates* permitem que uma classe chame métodos em outras sem exigir que esta outra classe seja derivada de um ancestral conhecido (SANT’ANNA, 2000).

No C# não existe a noção de ponteiros na memória. Isso quer dizer que não existe a eficiência dos ponteiros: muitos objetos são tratados por referência. As referências são “ponteiros domesticados”. Embora internamente elas sejam ponteiros, elas não podem apontar para locais arbitrários da memória. A memória não precisa ser liberada pelo programador pois existe um “*garbage collector”* que faz o serviço de limpar a memória evitando erros liberação da mesma. (SANT’ANNA, 2000).

Com os conceitos de *framework* e a linguagem de programação selecionada, é necessário a seleção de um IDE (*Integrated Development Environment*) para desenvolvimento do *framework*. Na seção a seguir é apresentado o Microsoft Visual Studio, a IDE selecionado para criação do *framework.*

2.3 Visual Studio

Segundo a Microsoft (2016), o Visual Studio é um conjunto completo de ferramentas para construção de aplicativos *desktops*, *web* e *mobile*. Nele é possível desenvolver em diversas linguagens como: C#, C++ (DEITEL,2006), Java Script (MORRISON,2008), Python (MENEZES,2014) e VB.Net (MANZANO,2015).

Na Figura 1, pode-se visualizar a IDE do Microsoft Visual Studio com um projeto aberto e janela do *solution explorer* para navegar nos arquivos do projeto.

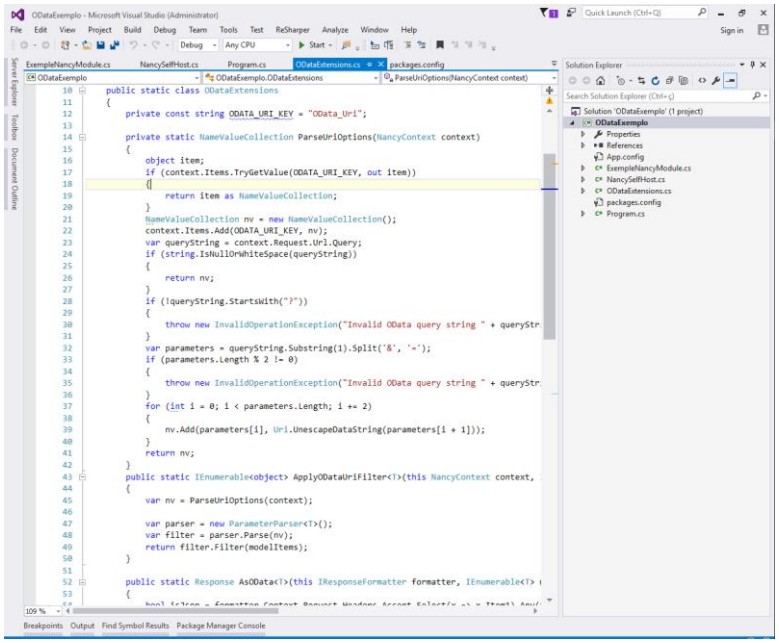

Figura 1 - Microsoft Visual Studio 2015

No Microsoft Visual Studio é possível criar vários tipos de sistemas e aplicativos como sistemas *web*, *sistemas desktops*, aplicativos para Windows *store*, jogos para clientes móveis e sistemas grandes e complexos.

Segundo a (MICROSOFT,2016), é possível à criação de jogos e aplicativos para Windows, Android (LECHETE,2013) e IOS (LECHETE,2016). Também permite a criação de

sites e serviços webs com base no ASP.NET (ABREU,2014), JQuery (SILVA,2013) e AngularJS (PEREIRA,2014). No Microsoft Visual Studio é possível também criar aplicativos para Microsoft Azure (TULLOCH,2013), Microsoft Office, Microsoft Sharepoint (MICROSOFT,2014) e Microsoft Kinect (CORDOSO,2013), além de aplicativos com uso intensivo de gráficos utilizando-se *DirectX*.

2.4 UML

A UML (*Unified Modeling Linguage*) é uma linguagem visual utilizada para modelar software baseados no paradigma de orientação à objeto. Ela surgiu da união de três métodos de modelagem diferentes, que são: o método de Booch, o método de OMT (*Object Modeling Tecnique*) e método de OOSE (*Object-Oriented Software Engineering*) (GUEDES, 2009, p. 17).

Contudo deve ficar bem claro que a UML não é uma linguagem de programação, ela é uma linguagem de modelagem que tem como objetivo auxiliar os engenheiros de software a definirem as características do sistema, os requisitos, sua estrutura lógica, dinâmica dos processos e a necessidade física que o sistema necessita para ser implantado (GUEDES, 2009, p. 17).

2.4.1 Diagrama de Classe

O diagrama de classe é um dos mais importantes diagramas que a UML possui. Sua principal função está em permitir a visualização das classes que comporão o sistema com seus respectivos atributos e métodos, além de demonstrar o relacionamento entre elas. (GUEDES, 2009, p. 101).

De acordo com Guedes (2009, p. 101), “Basicamente, o diagrama de classes é composto por suas classes e pelas associações existentes entre elas, ou seja, os relacionamentos entre as classes”. Na Figura 2, é apresentado um exemplo de um diagrama de classe.

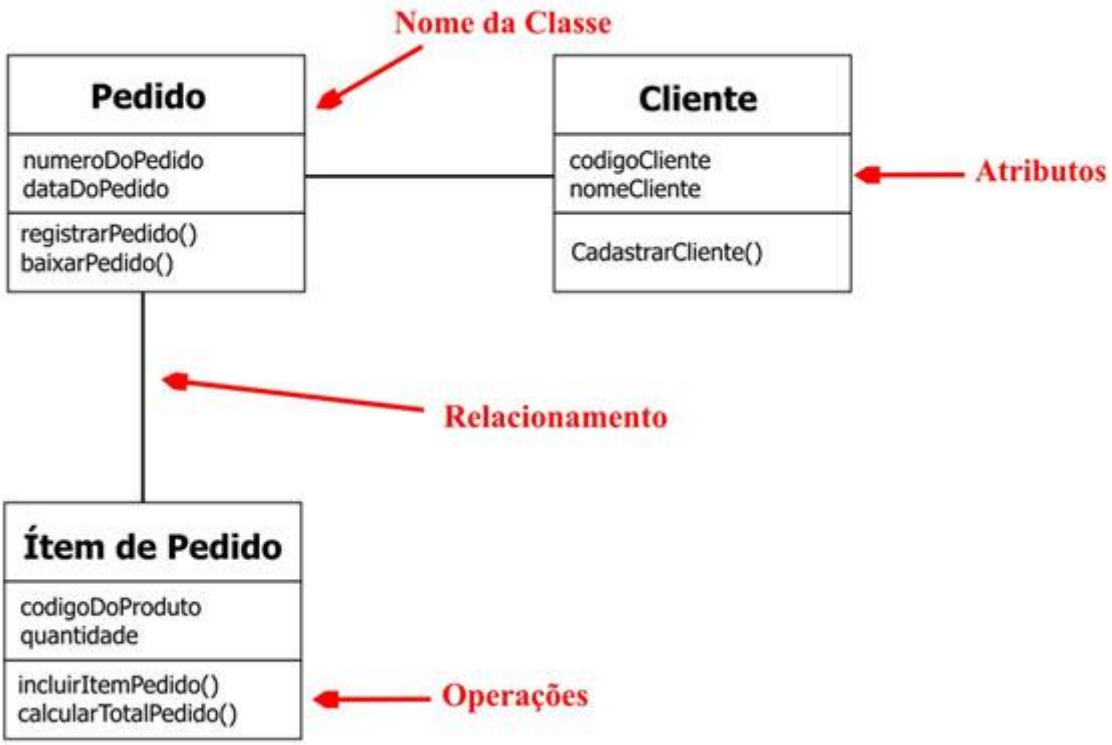

Figura 2 - Exemplo diagrama de classe

Na Figura 2, é possível verificar que o diagrama de classe tem como função permitir à visualização dos relacionamentos das classes. Nesse exemplo é apresentado o relacionamento da classe Pedido com Item de Pedido e Cliente, além disso ele demostrar os atributos de cada classe e os métodos que cada classe possui.

2.4.2 Diagrama de Atividades

De acordo com Guedes (2009, p. 277), “O diagrama de atividades era considerado um caso especial do antigo diagrama de gráfico de estados, chamado atualmente de máquina de estado”. A partir da UML 2.0, o diagrama de atividade passou a ser algo totalmente independente, deixando até de ser basear no diagrama de máquinas de estado, e se baseado no diagrama de redes de Petri (Guedes 2009, p. 277).

Guedes (2009, p.277) descreve a modelagem do diagrama de atividades como:

> "A Modelagem de atividade enfatiza a sequência e condições para coordenar comportamentos de baixo nível. Dessa forma, o diagrama de atividade é o diagrama com maior ênfase ao nível de algoritmo da UML e provavelmente um dos mais detalhista. "

O diagrama de atividade tem como finalidade a modelagem de atividades que podem ser um método ou um algoritmo, ou mesmo um processo completo. Ele também pode ser utilizado para modelagem organizacional, para engenharia de processos de negócio e *workflow*. O diagrama de atividades também é utilizado para modelagem de sistemas de informação para especificar processos ou nível de sistema (Guedes 2009, p. 277).

Na Figura 3, é apresentado um exemplo de um diagrama de atividade.

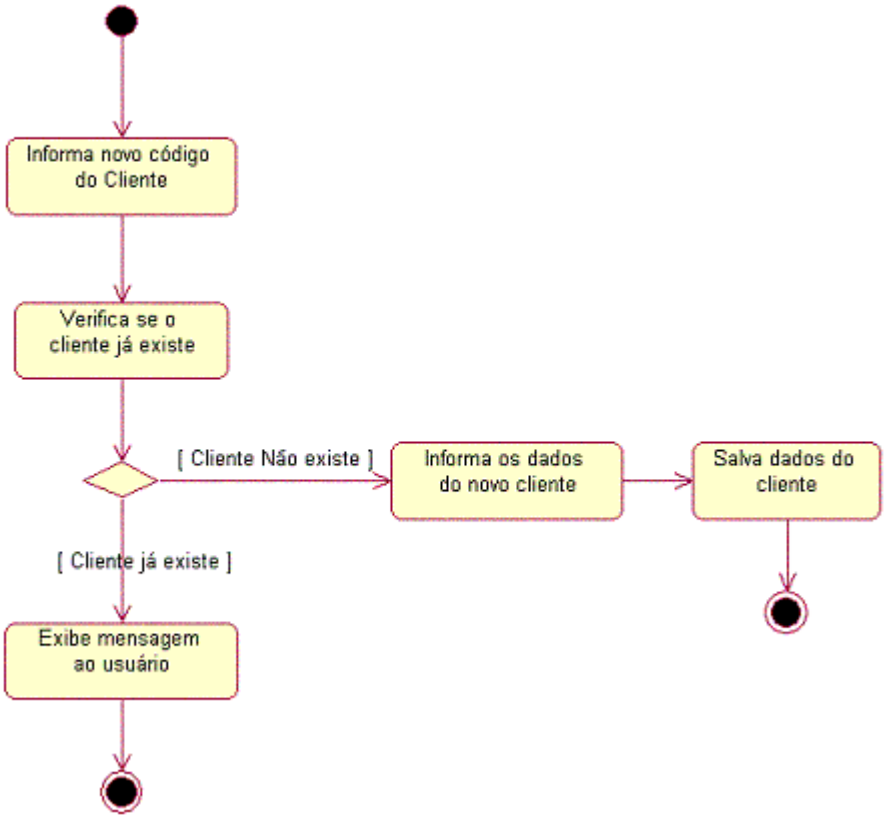

Figura 3 - Exemplo diagrama de atividade

Na Figura 3, é possível verificar que o diagrama de atividade modela o processo de um cadastro de cliente. Nesse exemplo deve-se informar um código para criar um novo cliente, após isso validar esse código verificando se ele já existe, caso exista retorna uma mensagem informando que o cliente já existente assim finalizando o fluxo do processo. Caso não exista o

código do cliente, ele deve informar os dados do novo cliente e salvar esses dados, finalizando assim o fluxo do processo.

2.5 Arquitetura em Camadas

A palavra arquitetura refere-se ao conceito de projetar e edificar o ambiente habitado pelo ser humano. Essa definição é claramente voltada a área de engenharia. No sentindo computacional do desenvolvimento de sistema ela tem relação com organização e divisão do *software* (FURGERI, 2015, p. 135).

De acordo com Furgeri (2015, p. 136), a definição de arquitetura na computação é "básica e que cada camada forneça serviços à outra camada imediatamente superior (atua como um provedor de serviços) e consumindo serviços na camada inferior (atua como um cliente) ".

2.5.1 Arquitetura cliente-servidor de "n" camadas

Arquitetura cliente-servidor de "n" camadas possui diversas variações. Trata-se da arquitetura mais usada e indicada para desenvolvimento de aplicações *web*. Nela as camadas podem estar divididas da seguinte forma: Camada do cliente, Camada de apresentação, Camada negócio e Camada de dados (FURGERI, 2015, p. 142).

De acordo com Furgeri (2015, p. 142), "A camada do Cliente basicamente é composta por *softwares* de navegador (*browser*) como: Internet Explorer, Chrome, Mozilla Firefox, etc". Já a camada de apresentação contém o servidor web responsável por disponibilizar todas as aplicações cliente (FURGERI, 2015, p. 142).

A camada de negócio é conhecida como a camada lógica sendo responsável por manter as regras de negócio da aplicação (FURGERI, 2015, p. 142). E segundo Furgeri (2015, p. 142), a camada de dados "é camada responsável pelo acesso e gerenciamento de todos os dados manipulados pelo sistema".

2.5.2 MVC

O modelo MVC (*Model-View-Controller*) é modelo de arquitetura de *software* composta por três camadas: modelo, visão e controle. Cada camada é responsável por estabelecer um conjunto de serviços para outras camadas (FURGERI, 2015, p. 142).

De acordo com Furgeri (2015, p. 143) "a ideia central em dividir o *software* em camadas é facilitar o reaproveitamento de código, o desenvolvimento e a manutenção".

Furgeri (2015, p 143) descreve a camada modelo como: "A camada modelo é responsável por manter os dados da aplicação, gerenciando o acesso a esses dados, que normalmente são requisitados pela camada visão. "

A camada visão é responsável por apresentar as informações para o usuário. Esses dados podem ser apresentados de diversas maneiras como em telas, uma tabela ou um diagrama (FURGERI, 2015, p. 143).

A camada controle é responsável por fazer comunicação entre a camada visão e camada modelo (FURGERI, 2015, p. 143). A seguir, a Figura 4 apresenta a arquitetura *MVC*.

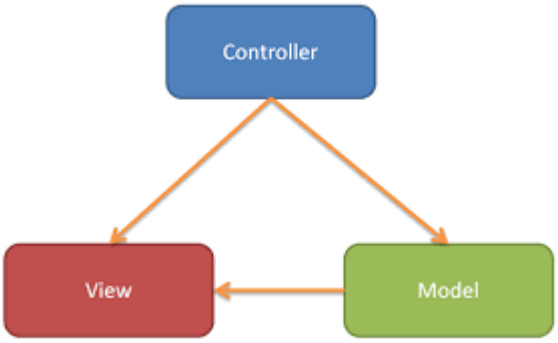

Figura 4 - O modelo MVC

2.6 ADO.NET *Entity Framework*

De acordo com Microsoft (2015), "o ADO.NET *Entity Framework* é um *framework* do tipo ORM (Object/Relational Mapping) que permite aos desenvolvedores trabalhar com dados relacionais como objetos de domínio específico".

Com o *Entity Framework*, “os desenvolvedores podem lançar consultas usando LINQ, e depois recuperar e manipular dados como objetos fortemente tipificados” (Microsoft, 2015).

O *Entity Framework* funciona com diversos servidores de SGDBs como, Microsoft SQLServer, Oracle e IBM DB2. O *Entity Framework* inclui um sofisticado mecanismo de mapeamento que pode lidar com esquemas reais de banco de dados (MICROSOFT, 2015). A seguir, a Figura *5* apresenta a arquitetura do *Entity Framework.*

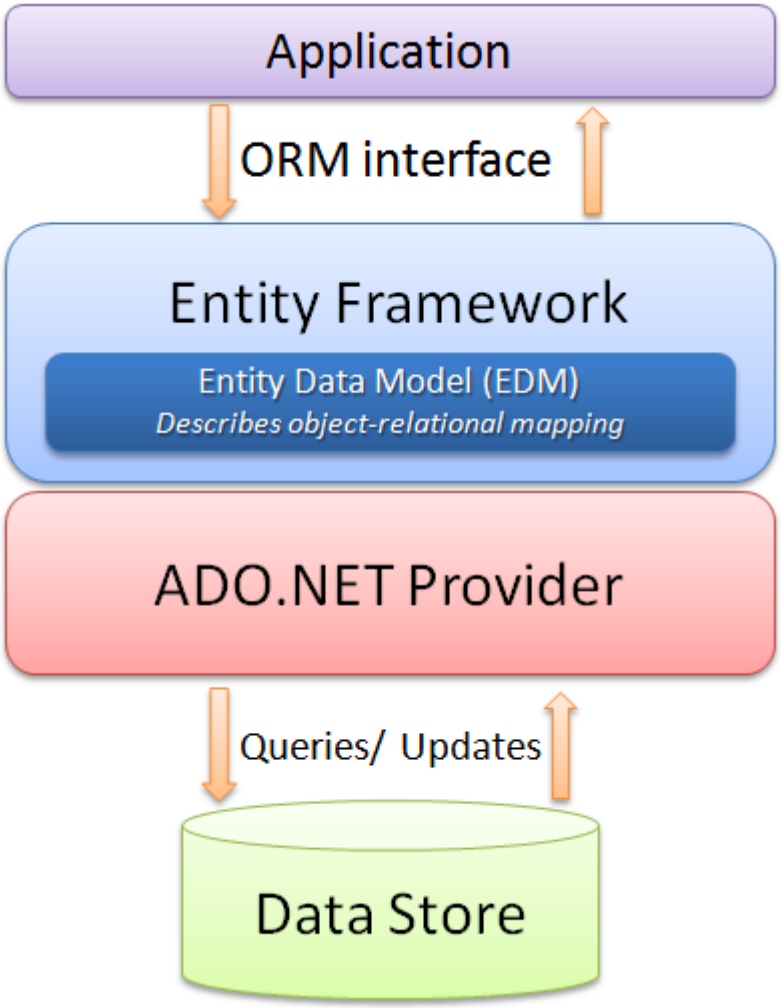

Figura 5 - Arquitetura do *Entity Framework*

O *Entity Framework* é baseado no modelo de provedor ADO.NET, portanto aplicações que utilizam esse mesmo provedor podem utilizar facilmente o *Entity Framework* como modelo de programação devido sua similaridade com o desenvolvimento ADO.NET (MICROSOFT,2015).

3 DESENVOLVIMENTO

3.1 Visão Geral

O desenvolvimento do *framework* foi realizado na linguagem de programação C# versão 6.0 por apresentar as características que facilitam o desenvolvimento de aplicações genéricas. Uma dessas características é o "*Reflection*" que possibilita trabalhar com os meta-dados das propriedades. O *Reflection* pode ser utilizado com a biblioteca *System.Reflection*.

O C# ainda apresenta uma palavra chave chamada *dynamic* que possibilita chamar métodos e propriedades de um objeto qualquer, sem conhecer o seu tipo. Isso é conhecido como *late-binding*, ou seja, realizar a associação do método ou propriedade apenas quando o mesmo for chamado. Ela apresenta ainda atualização de *data annotations* que são notações em classes e atributos que pode ser utilizada para validação, mapeamento de objetos entre outras funções.

Para desenvolver utilizando o C# é necessário uma IDE para desenvolvimento. Existem inúmeras IDEs como: MonoDevelop, Microsoft Visual Studio, que oferecem suporte ao C#, contudo foi utilizado o Microsoft Visual Studio *community* 2015. Ele foi escolhido por ter como linguagem padrão o C# e por apresentar vários recursos que facilitam o desenvolvimento como o auto completar que facilita para o desenvolvedor não ter que decorar as tag da linguagem, e um verificar que valida código em tempo de escrita, evitando possíveis erros de escrita do código.

3.2 O *Framework*

O *framework* foi criado para padronizar e agilizar o processo de desenvolvimento de aplicações utilizando a linguagem de programação C#. Contudo, apenas esse fato não é o suficiente para a criação de um novo *framework* pois existem vários no mercado como: ADO.NET *Entity Framework*, *NHibernete*, que fazem o mesmo.

O *framework* foi desenvolvido para criar um padrão para desenvolvimento utilizando o conceito de arquitetura de "n" camadas. O mesmo irá fornecer diversos repositórios para diferentes camadas da aplicação, tirando assim a necessidade de criar um repositório para cada projeto que for desenvolvido.

Além disso, o *framework* possui uma parte construído especialmente para trabalhar com projetos ASP.NET MVC, que trabalha com arquitetura MVC. Nesses projetos, o *framework* tem como finalidade auxiliar a comunicação dos *controllers* com as *views*. Em projetos que utilizarem o ASP.NET MVC e uma arquitetura de "n" camadas, o *framework* já está preparado para fazer conversões de objetos do mapeamento do banco de dados para objetos utilizados nas *views*.

O *framework* foi desenvolvido diferente de outros *frameworks* presentes no mercado, possui outras finalidades além da persistência de dados. O *framework* desenvolvido tem com princípio agilizar o desenvolvimento por apresentar diferentes repositórios de código para diferentes camadas de um projeto.

O *framework* também tem como finalidade facilitar a conexão com vários bancos de dados para aplicações que necessitam conectar em mais de uma base de dados. Para realizar tais conexões em diferentes bancos de dados, foi elaborado uma *data annotation* que deve ser inserida nas classes que irão comunicar com banco dados, pois essas anotações irão armazenar as informações necessárias para estabelecer tais conexões.

Na Figura 6, é apresentado o diagrama de classe do *framework*, demostrando as ligações que cada classe possui uma com a outra, demostrando as classes que são herdadas por outras classes e as classes que implementam as interfaces no *framework.*

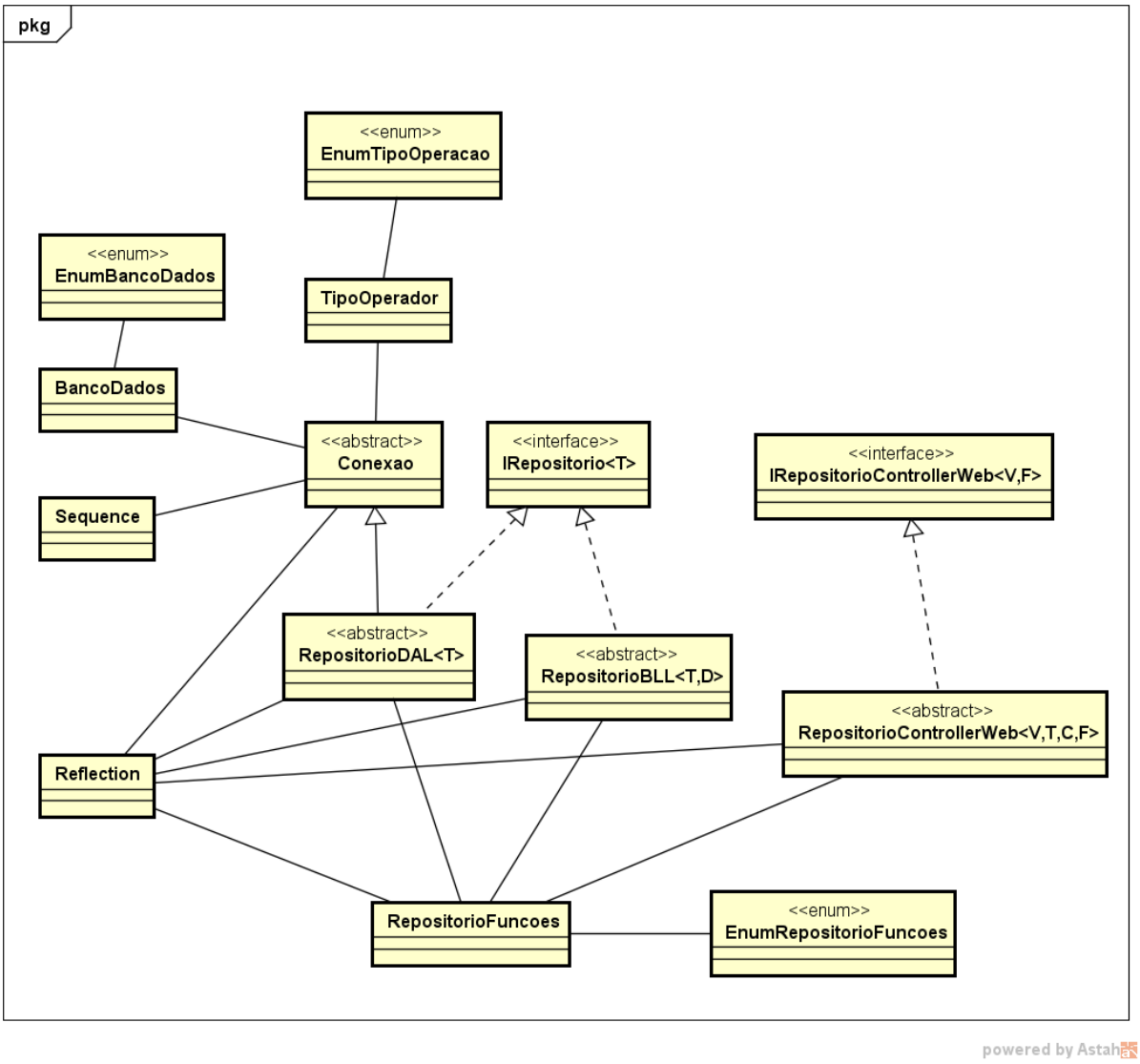

Figura 6 - Diagrama de classe do *framework*

Na Figura 7, é apresentado o diagrama de atividade do *framework* demostrando o fluxo de funcionamento dele, e apresenta as entradas de dados e possíveis saídas que pode ocorrer no funcionamento do *framework.*

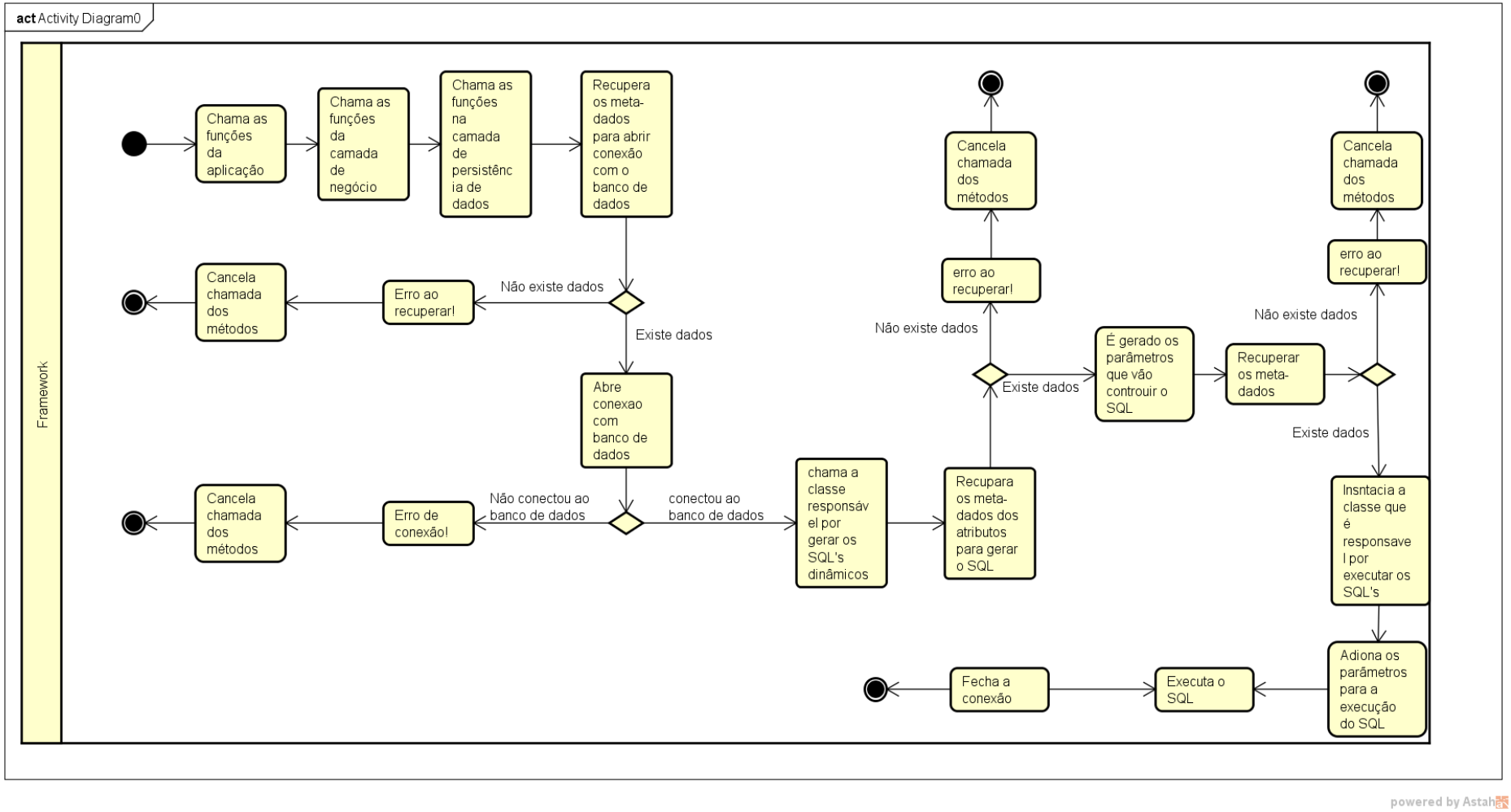

Figura 7 - Diagrama de atividade do *framework*.

3.3 Estrutura do projeto

Antes de desenvolver um *framework*, deve-se pensar na estrutura do projeto. A separação das classes e a organização das heranças é de grande importância para o projeto ter uma estrutura organizada e de fácil entendimento.

A estrutura do *framework* apresenta uma organização por pastas. Essas pastas foram segmentadas de acordo com a necessidade e a responsabilidade que suas classes terão no projeto. A estrutura do projeto final pode ser vista na Figura *8*.

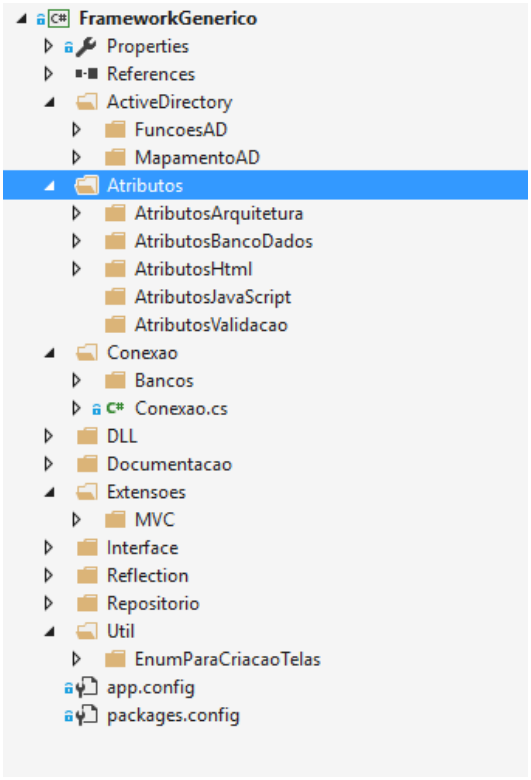

Figura 8 - Estrutura do projeto do framework

3.3.1 Estrutura da Conexão

Na elaboração da classe de conexão, foi criada uma estrutura que suportará vários bancos de dados. Nela foi utilizada objetos dinâmicos que terão apenas um tipo específico na execução, facilitando assim a implementação dos bancos de dados.

Para cada banco de dados foi criado uma classe própria para inserção do código e dependências, pois cada banco de dados tem sua própria DLL (*Dynamic-Link Library*) para realização da comunicação entre a linguagem C# e o banco de dados. Os códigos presentes nelas são similares. Segue, a seguir, na Figura 9, o código da classe que irá controlar os métodos do banco de dados Oracle.

```
internal class Oracle
{

    1 reference
    public static OracleConnection AbrirConexao(string conexao)
    {
        try
        {
            OracleConnection connection = new OracleConnection(conexao);
            connection.Open();
            return connection;
        }
        catch (Exception ex)
        {
            throw new Exception("Erro ao Instanciar Conexão" + ex.Message);
        }
    }


    1 reference
    public static OracleCommand InstanciarCommand(dynamic connection, dynamic transaction)
    {
        try
        {
            return new OracleCommand { Connection = connection, Transaction = transaction };
        }
        catch (Exception ex)
        {
            throw new Exception("Erro ao Instanciar Conexão" + ex.Message);
        }
    }
}
```

Figura 9 - Classe para trabalha com banco de dados Oracle

Como apresentado o na Figura 9, a classe é do tipo *internal*, ou seja, essa classe dever ser acessada apenas por outras classes do *framework.* A classe possui apenas dois métodos que devem ser implementados: um método para abrir a conexão e outro para instanciar o *Command*, que é responsável por executar os SQLs. Devido a esses métodos é que o *framework* deve ter

uma classe para cada banco, pois são nesses métodos que são realizados as conexões com o banco de dados e são utilizados as DLLs de cada banco de dados.

Com essa separação, o *framework* retira a necessidade da presença de todas as DLLs nos projetos, pois o projeto que estiver utilizando o *framework* terá a necessidade apenas de inserir a DLL do banco de dados utilizado. Por exemplo, um projeto utilizando o *framework* proposto que utilizar banco de dados Oracle deve ter apenas a DLL padrão do Oracle.

Contudo, a cada banco de dados adicionado no *framework* deve-se implementar uma condição na classe de conexão para que mesma consiga chamar e realizar operações nesse novo banco de dados, como exemplificado na Figura 10.

```
switch (Reflection.Reflection.RecuperarBancoDadosUtilizado(GetType()))
{
    case EnumBancosDados.Oracle:
        return Bancos.Oracle.AbrirConexao(ConnectionString());
    case EnumBancosDados.MySql:
        return Bancos.MySql.AbrirConexao(ConnectionString());
    case EnumBancosDados.PostgreSql:
        return Bancos.PostgreSql.AbrirConexao(ConnectionString());
    case EnumBancosDados.SqlServer:
        return Bancos.SqlServer.AbrirConexao(ConnectionString());
    default:
        throw new Exception("Informe em Qual banco deve ser Conectado");
}
```

Figura 10 - Validação de qual banco de dados será utilizado

3.4 Reflection

Na elaboração desse *framework* foi utilizado o conceito de *reflection*, que é a utilização de meta-dados de atributos e classes. Para utilizar os recursos que a *reflection* possibilita foi criado uma classe chamada *Reflection*. Nela está todos os métodos que utiliza *reflection* no *framework*. Essa classe tem como principal funcionalidade tratar e recuperar todos os valores e eventos relacionados com as *data annotation* das classes e atributos.

Para que essa classe não ficasse com uma estrutura de difícil entendimento, por implementar diversos métodos, foi utilizado o conceito de *region*[*]. Um exemplo de *Region* é apresentado na Figura 11.

[*] *Region* é uma tag do C# que possibilita a divisão do código para melhor visualização e organização

```
internal class Reflection
{

    /// <summary>
    /// Alterar a Cultura Conforme a Necessidade
    /// </summary>
    public static CultureInfo Culture = new CultureInfo("pt-BR");

    Operações de get set

    Banco de Dados

    Operações no Banco de Dados

    #region Instanciar e Invocar Metodos

    2 references
    public static Assembly RecuperarAssembly(string projeto)
    {
        try
        {
            return Assembly.Load(projeto);
        }
        catch (Exception ex)
        {
            throw new Exception("Erro ao recuperar Assembly: " + ex.Message);
        }
    }
```

Figura 11 - Classe *Reflection*

Nessa classe *Reflection* existem alguns métodos importantes, como por exemplo, recuperar o *assembly* de um projeto pelo nome do projeto, como pode ser visto na Figura 11. Também é possível recuperar o tipo do atributo e atribuir valores dinamicamente a algum atributo. Alguns desses métodos são apresentados na Figura 12.

```
5 references
public static void SetarValorAtributo(PropertyInfo propertyInfo, Object clase, dynamic valor)
{
    try
    {
        propertyInfo.SetValue(clase, valor.ToString() != "" ? Convert.ChangeType(valor, propertyInfo.PropertyType, Culture) : Activator.CreateInstance(propertyInfo.PropertyType));
    }
    catch (Exception ex)
    {
        throw new Exception("Erro ao Setar valor no Atributo: " + ex.Message);
    }
}

1 reference
public static string RecuperarTipoObjeto(object obj)
{
    try
    {
        return obj.GetType().Name;
    }
    catch (Exception ex)
    {
        throw new Exception("Erro ao Recuperar Tipo do Objeto: " + ex.Message);
    }

}
```

Figura 12 - Métodos da classe *Reflection*

3.5 Repositório

No desenvolvimento do *framework* foi criado uma pasta com nome de Repositório. Ela foi criada para armazenar quatro classes do *framework*, são elas: RepositorioBLL, RepositorioDAL, RepositorioControllerWeb e RepositorioFuncoes. Essas quatro classes são responsáveis por controlar as funções do *framework*.

3.5.1 RepositorioDAL

A classe RepositorioDAL é responsável pela persistência de dados com os bancos de dados implementados no *framework*. Ela comunica diretamente com a classe de conexão, com a classe de *Reflection* e com a classe RepositorioFuncoes. A classe de conexão faz todo trabalho de conectar e abrir transação com banco de dados. Já a classe de *Reflection* é a classe que vai recuperar os meta-dados necessários, como por exemplo, recuperar a *data annotation* que armazena o nome da tabela à qual será executado o método. Já o RepositorioFuncoes é responsável por gerar os SQLs dinâmicos.

Na Figura 13 é apresentado o método de deletar, e na Figura 14 é apresentado o método de consultar todos com filtro.

```
4 references
public virtual void Deletar(T t)
{
    try
    {
        using (Connection = OpenConnection())
        {
            Sql = "";
            AbrirTransaction();
            Command = InstanciarCommand();
            PrencherSqlDictionaryCommand(RepositorioFuncoes.GerarFuncoes(EnumRepositorioFuncoes.Deletar, t, Bind, null));
            Command.CommandText = Sql;
            Command.ExecuteNonQuery();
            CommmitTransaction();
        }
    }
    catch (Exception ex)
    {
        try
        {
            RollbackTransaction();
            throw new Exception("Erro ao executar o método Deletar: " + ex.Message);
        }
        catch (Exception exRollback)
        {
            throw new Exception("Erro ao exucutar Rollback: " + exRollback.Message);
        }
    }
    finally
    {
        CloseConnection();
    }
}
```

Figura 13 - Método para deletar no RepositorioDAL

```
4 references
public virtual List<T> ConsultarTodosComFiltro(dynamic filtros, long? limit, long? offset)
{

    try
    {
        Sql = "";
        using (Connection = OpenConnection())
        {
            dynamic objeto = GetObjeto();
            Command = InstanciarCommand();
            Sql = "select * from " + (string.IsNullOrEmpty(Reflection.Reflection.RecuperarSchemaTabela(objeto.GetType())) ? "" :
                Reflection.Reflection.RecuperarSchemaTabela(objeto.GetType()) + ".") + Reflection.Reflection.RecuperarNomeTabela(objeto.GetType());
            if (filtros != null)
                PrencherSqlDictionaryCommand(RepositorioFuncoes.GerarFuncoes(EnumRepositorioFuncoes.FiltroSql, filtros, Bind, null));
            if (limit != null && offset != null)
                MontaPaginacao((long)limit, (long)offset);
            Command.CommandText = Sql;
            var DataReader = Command.ExecuteReader();
            List<T> list = new List<T>();
            while (DataReader.Read())
            {
                var obj = LerDataReader(GetObjeto(), DataReader);
                list.Add(obj);
            }
            DataReader.Close();
            return list;
        }
    }
    catch (Exception ex)
    {
        throw new Exception("Erro ao executar o método ConsultarTodosComFiltro: " + ex.Message);
    }
    finally
    {
        CloseConnection();
    }
}
```

Figura 14 - Método para consultar todos com filtro na classe RepositorioDAL

3.5.2 RepositorioBLL

A classe RepositorioBLL tem como objetivo ser a responsável pela regra de negócio do *framework*. É nela que é realizada todas as operações de validações necessárias. Ela também é responsável por chamar os métodos da classe RepositorioDAL. na Figura 15 é apresentado o método de atualizar da classe RepositorioBLL.

```
1 reference
protected abstract void ValidarAntesAtualizar(T t);
1 reference
protected abstract void ValidarDepoisAtualizar(T t);

4 references
public virtual void Atualizar(T t)
{
    try
    {
        ValidarAntesAtualizar(t);
        GetDal().Atualizar(t);
        ValidarDepoisAtualizar(t);
    }
    catch (Exception ex)
    {
        throw new Exception("Erro ao executar Método Atualizar: " + ex.Message);
    }
}
```

Figura 15 - Método atualizar da classe RepositorioBLL

Na Figura 15 podemos observar que basicamente o método realiza a chamada de 3 outros métodos da classe RepositorioDAL, que são: ValidarAntesAtualizar, ValidarDepoisAtualizar e o método Atualizar. Os métodos ValidarAntesAtualizar e ValidarDepoisAtualizar são métodos abstratos que devem ser implementados na classe que herdar o RepositorioBLL.

No RepositorioBLL existe um método na qual se cria a instância de alguma classe, normalmente será utilizado esse método para criar a instância da classe que foi informado na dependência do RepositorioBLL. O objetivo dessa instância é executar os métodos do RepositorioDAL, como apresentado na Figura 16.

```
9 references
protected D GetDal()
{
    try
    {
        return Activator.CreateInstance<D>();
    }
    catch (Exception ex)
    {
        throw new Exception("Erro a intanciar objeto genérico: " + ex.Message);
    }
}
```

Figura 16 - Método para criar uma instancia de uma classe

3.5.3 RepositorioControllerWeb

A classe RepositorioControllerWeb deve ser utilizada nos *controllers* de projetos ASP.NET MVC. Ela tem como responsabilidade chamar os métodos da camada de negócio e comunicar diretamente com as *views* do projeto APS.NET MVC. O RepositorioControllerWeb deve ser utilizado apenas quando o projeto utilizar as camadas BLL (*Business Logic Layer*) e DAL (*Data Access Layer*).

Na Figura 17 é apresentado o método Atualizar do RepossitorioControllerWeb. Nesse método existe uma validação padrão do APS.NET MVC que verifica se o objeto informado é válido.

No método apresentado na Figura 17 é possível verificar que existe uma chamada na classe RepositorioFuncoes que realiza a conversão do objeto ViewModel para objeto Model. Esse método é chamado pois o objeto ViewModel é criado para tratar informações necessárias na *view* sendo assim ela é diferente do objeto Model que é a réplica de uma tabela do banco de dados.

```
1 reference
public virtual ActionResult Atualizar(V v)
{
    try
    {
        if (!ModelState.IsValid)
            return View("Editar");
        dynamic Model = RepositorioFuncoes.ConverteObjetoToObjeto(v, GetT(),
            EnumTipoConvercoes.DataAnnotations);
        GetC().Atualizar(Model);
        TempData.Add("Sucesso", "Dados Atualizados com sucesso!");
        return RedirectToAction("Visualizar", MontaRota(v));
    }
    catch (Exception ex)
    {
        TempData.Add("Erro", "Erro ao Atualizar: " + ex.Message);
        return View("Editar");
    }
}
```

Figura 17 - Método atualizar do RepositorioControllerWeb

3.5.4 RepositorioFuncoes

A classe RepositorioFuncoes foi desenvolvido pensando em tirar algumas responsabilidades dos outros repositórios como o método para gerar SQL. O método de gerarSQL foi criado na classe RepositorioFuncoes, pois se existisse outro repositório que trabalhasse com o banco de dados não seria necessário refatorar* o código para conseguir reaproveitar o mesmo.

Além disso, existe algumas validações com base nas *data annotati*on que são executadas no RepositorioFuncoes assim, basicamente o RepositorioFuncoes tem a responsabilidade de agrupar métodos que possam ser executados em diferentes repositórios.

Na Figura 18 é apresentado o método para fazer uma validação básica verificando se o SQL está vazio ou nulo. Essa validação existe, pois, se o método não conseguir recuperar os dados necessários para criar o SQL, o *framework* não irar executar nenhuma função no banco de dados.

* Refatorar é o processo de melhorar a estrutura do código preservando seu comportamento externo.

```
4 references
protected static void ValidaSqlDictionaryCommand()
{
    try
    {
        if (string.IsNullOrEmpty(Sql))
            throw new Exception("Sql não pode ser null ou vazio");
        if (DictionaryCommand.Count <= 0)
            throw new Exception("DictionaryCommand não pode estar vazio");
    }
    catch (Exception ex)
    {
        throw new Exception("Erro ao Validar DictionaryCommand: " + ex.Message);
    }
}
```

Figura 18 - Método para validar SQL

4 RESULTADOS

Para demonstrar os resultados do *framework* foi elaborado um protótipo. No protótipo é possível fazer quatro cadastros diferentes que são: cadastro de tipo de produto, categoria, produto e categoria x produto. Nessa aplicação é possível verificar o funcionamento do *framework* tanto em cadastros simples quanto em cadastros com chave estrangeira* e entidades fracas*.

Para utilizar o *framework* é necessário, realizar a instalação do mesmo no projeto, ou inserir a DLL do *framework* direto no projeto.

Esse projeto foi estruturado em camadas, são elas: DTO (*Data Transfer Object*) camada utilizada para fazer a representação do banco de dados, DAL (*Data Access Layer*) camada responsável pela comunicação com o banco de dados, BLL (*Business Logic Layer*) camada responsável pela regra de negócio e projeto WEB que é camada de interação com usuário. Na Figura 19 é apresentada a estrutura do projeto.

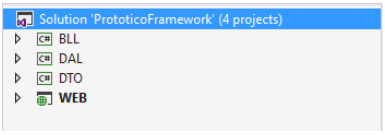

Figura 19 - Estrutura do projeto

4.1 Mapeamento da Aplicação

Para utilizar o *framework* é necessário o mapeamento das classes que são a representação do banco de dados. Sem esse mapeamento o *framework* não conseguiria realizar suas funções corretamente. Portanto, esse é um dos pontos mais importantes na utilização do *framework*. Na Figura 20 é apresentado o mapeamento da classe categoria.

* **Chave estrangeira** ou **chave externa** se refere ao tipo de relacionamento entre distintas tabelas de dados do banco de dados.

* Entidade fraca é uma entidade que não possui existência própria (sua existência depende da existência de outra entidade) ou que para ser identificada depende da identificação de outra entidade.

```
[Table("Categoria")]
24 references
public class Categoria
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    8 references | 0 exceptions
    public int IdCategoria { get; set; }
    5 references | 0 exceptions
    public string NomeCategoria { get; set; }
}
```

Figura 20 - Mapeamento da Classe Categoria

Como pode ser observado na Figura 20, para realizar o mapeamento da base de dados utiliza-se as *data annotation*. Nessa classe foram utilizadas a seguintes *data annotations*: *Table*, *Key*, *DatabaseGenerated*, *Column*. A *data annotation Table* é utilizada para informar o nome da tabela do banco de dados. A *Key* é utilizada para informar que o atributo é uma chave primária da tabela no banco de dados. A *data annotation Column* é utilizada para informar o nome do atributo no banco de dados e a *data annotation DatabaseGenerated* é utilizada para informar que a chave primária *dessa tabela é auto incrementada.

Esse mapeamento deve ser realizado em todas as classes que fazem a representação do banco de dados. No caso desse projeto, todas as classes que estão no projeto DTO devem ser esse mapeadas.

4.2 Utilização dos Repositórios

Após realizar o mapeamento deve-se realizar a herança nas classes que irão comunicar com banco de dados. Para isso deve-se utilizar o RepositorioDAL que, como foi apresentado anteriormente, é a classe responsável pela comunicação com banco de dados. Figura 21 é apresentado a classe CategoriaDAL.

```
[BancoDados(EnumBancosDados.SqlServer, "strConexao")]
1 reference
public class CategoriaDAL  : RepositorioDAL<Categoria>
{
}
```

Figura 21 - Classe CategoriaDAL

* Atributo ou combinação de atributos que possuem a propriedade de identificar de forma única uma linha da tabela

Como apresentado na Figura 21, a classe categoriaDAL possui uma *data annotation* chamada de BancoDados. Nessa *data annotation* define-se qual banco de dados essa classe utiliza e o nome da *string* de conexão utiliza para conectar nesse banco. Quando realiza-se a herança, é necessário informar a classe com mapeamento do banco de dados, que nesse caso, é a classe Categoria.

Em todas as classes da camada DAL deve-se definir à *data annatation* BancoDados, pois é com ela que o *framework* consegue identificar o banco de dados a ser utilizado e buscar a conexão no arquivo de configuração do projeto.

Logo após estruturar a camada DAL, fazendo todas configurações necessárias nas classes que pertencem, é necessária a realização da herança nas classes que serão responsáveis pela regra de negócio do sistema. Nesse caso deve-se utilizar o ReposiotiroBLL, como apresentado na Figura 22.

```
2 references
public class CategoriaBLL : RepositorioBLL<Categoria,CategoriaDAL>
{

}
```

Figura 22 - Classe CategoriaBLL

Na herança do RepositórioBLL é necessário informar a classe do mapeamento do banco de dados como na herança da RepositórioDAL. Contudo, quando herda-se de RepositórioBLL é necessário também informar uma classe da camada DAL. Isso é necessário pois a camada BLL tem reponsabilidade de comunicar diretamente com a camada DAL.

Realizado o processo de herança nas classes, que são responsáveis pela regra de negócio do sistema, deve-se realizar a herança nas classes *controllers*, que são criadas nos projetos ASP.NET MVC. Nesse caso será utilizado o RepositorioControllerWeb, que é responsável por fazer comunicação entre as *views* e a camada de negócio do sistema. A herança da classe CategoriaController é apresentado na Figura 23.

```
0 references
public class CategoriaController : RepositorioControllerWeb<ModelCategoria, Categoria, CategoriaBLL, FiltroCategoria>
{

}
```

Figura 23 - Classe CategoriaController

Já na herança do RepositorioControllerWeb também é necessário informar a classe que representa o banco de dados. Nessa herança deve-se passar uma classe da camada BLL, pois a

camada web não deve comunicar diretamente com camada DAL. Nessa mesma herança também é necessário informar uma classe que conterá os valores para filtro nas consultas e uma classe que representa os objetos que estão presentes na tela.

4.3 Telas Geradas

Para o desenvolvimento das telas foi utilizado o ASP.NET MVC, que é um padrão de desenvolvimento web da Microsoft. Para cada cadastro no protótipo foi criado quatro telas: tela de novo, tela de atualizar, a tela de visualizar e a tela de visualizar todos. Com exceção da tela de associação de categoria com produto, que possui apenas duas telas: a tela de associação de categoria com produto e a tela de visualizar todos.

As telas foram criar dessa maneira para demostrar o comportamento do *framework* quando o mesmo precisa fazer transição entre paginas

Como as telas de produto e tipo produto têm o mesmo padrão de categoria, a seguir será apresentado somente as telas da classe Categoria. A tela de atualizar categoria é apresentada na Figura 24, a tela de visualizar categoria é apresentada na Figura 25, a tela de nova categoria é apresentada na Figura 26 e a tela de visualizar todas as categorias é apresentada na Figura 27.

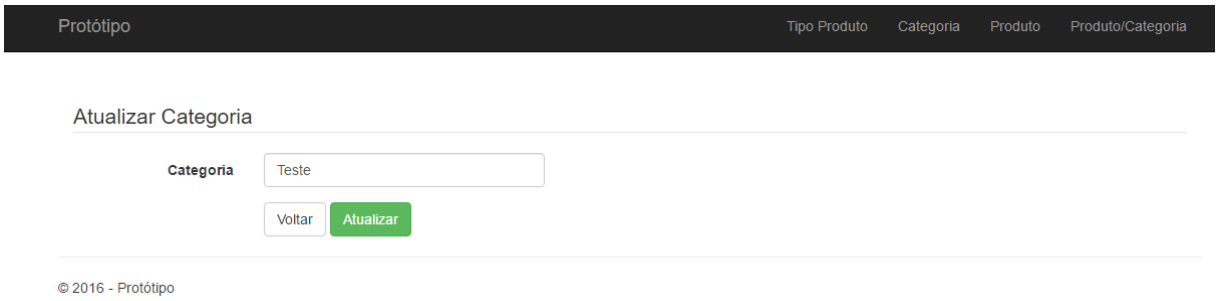

Figura 24 - Tela de atualizar categoria

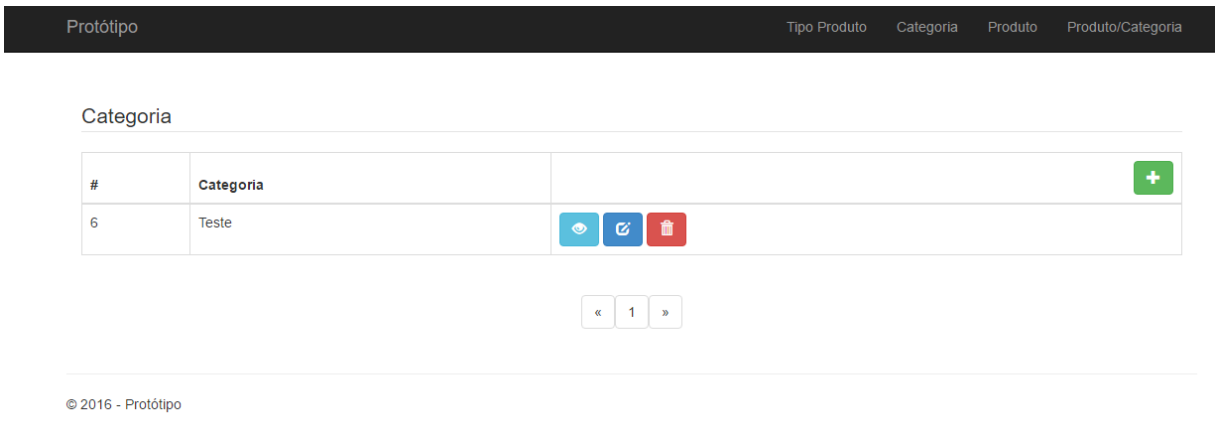

Figura 25 - Tela de visualizar todas as categorias

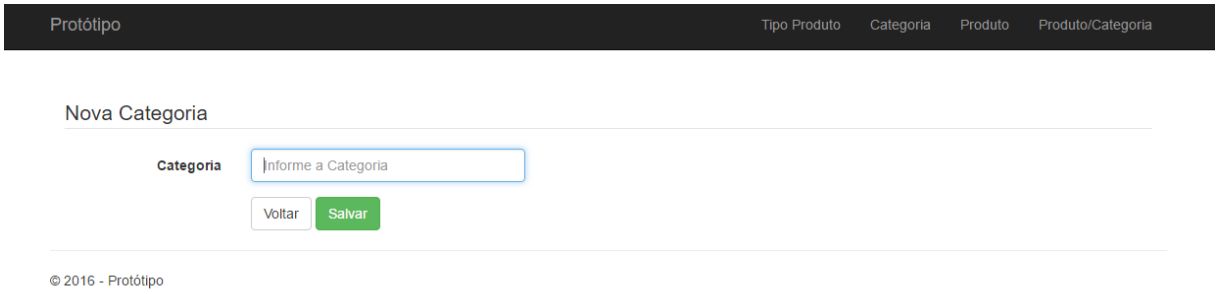

Figura 26 - Tela de nova categoria

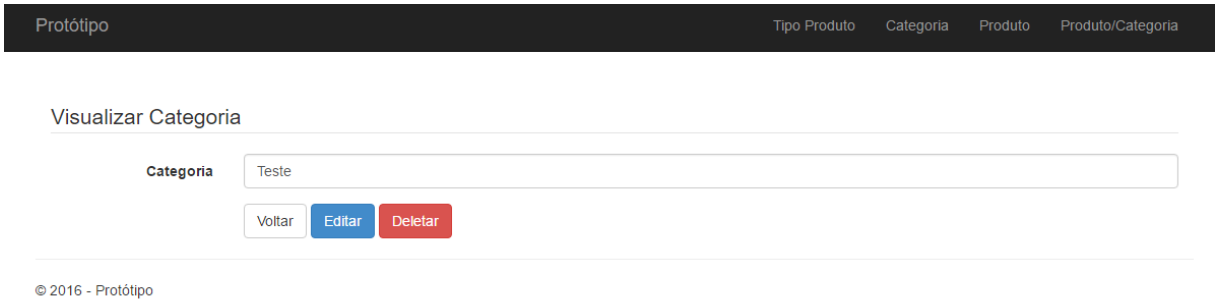

Figura 27 - Tela de visualizar categoria

As telas de categoria, tipo produto e produto têm o mesmo formato. Sendo assim as mesmas não foram apresentadas no texto. A seguir, na Figura 28, é apresentada a tela de associar categoria a produto e na Figura 29 é apresentada a tela de visualizar produtos associados a uma categoria.

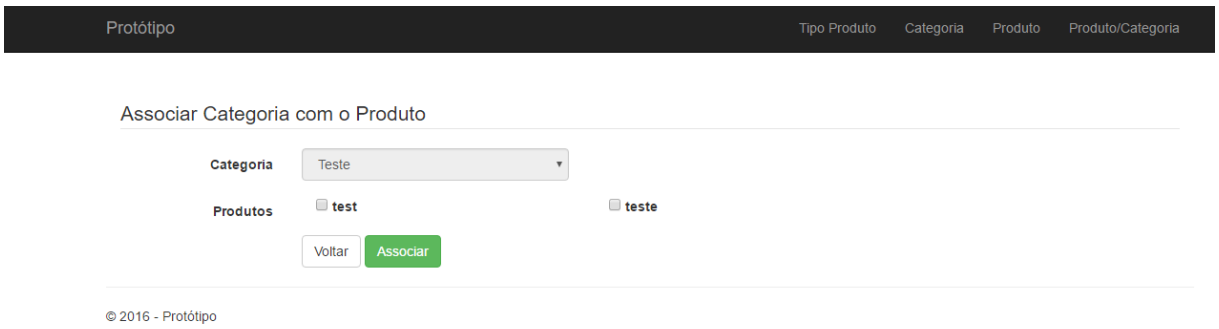

Figura 28 - Tela de associar categoria aos produtos

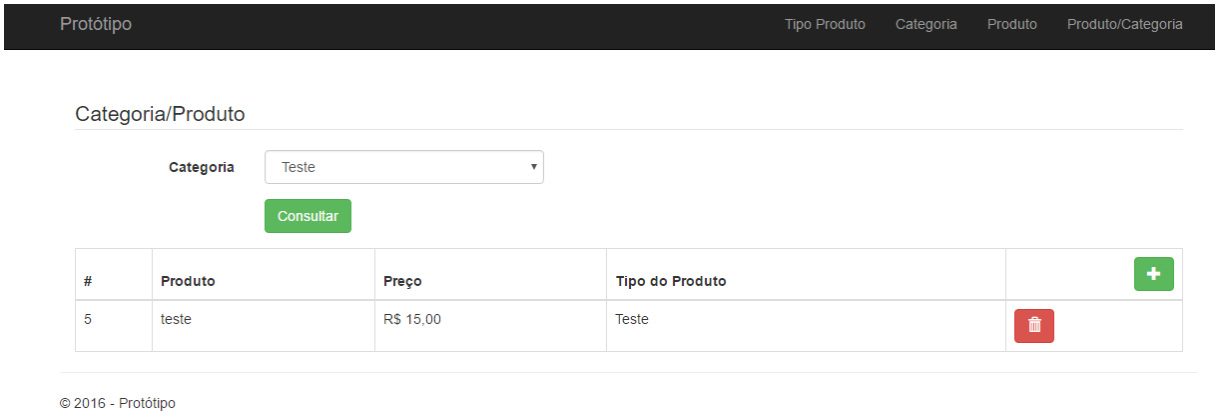

Figura 29 - Tela de visualizar todos produtos associados a uma categoria

4.4 SQL Gerados

Conforme exposto nos objetivos desse trabalho, as SQLs geradas pelo *framework* devem ser simples.

Para demostrar as SQLs geradas foi executado as quatro operações básica, que são: salvar, atualizar, deletar e consultar. Essas operações foram realizadas utilizando a classe Categoria. A seguir, na Figura 30, é apresentada a SQL para consultar todas Categorias com paginação, utilizando o Banco de Dados no SQLServer.

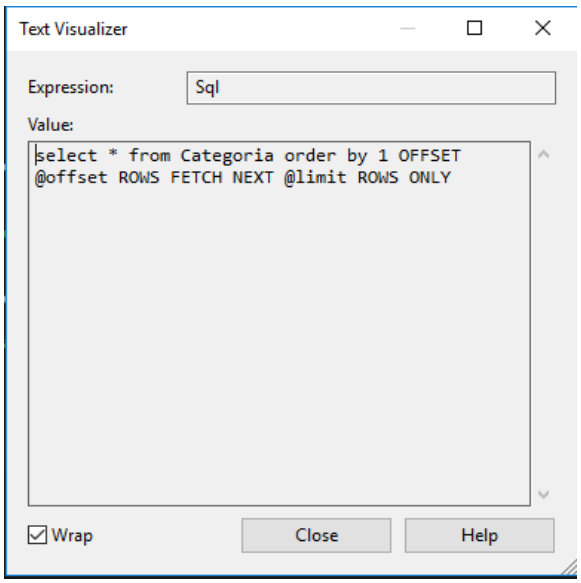

Figura 30 - SQL consultar todos com paginação

Na Figura 31 é apresentada a SQL gerada pelo *framework* para executar a função de *delete* na tabela Categoria sendo a condição para executar a remoção é a chave primária da tabela. Na Figura 32 é apresentada a SQL que executa a função de *update* na tabela Categoria, como na SQL para remoção, a condição para realizar essa operação é a chave primária da tabela. E na Figura 33 é apresentada a SQL de inserção de registro na tabela Categoria.

Todas essas SQLs foram geradas de forma dinâmica pelo *framework* a partir das *data annotation* presentes nas classes que representam o banco de dados.

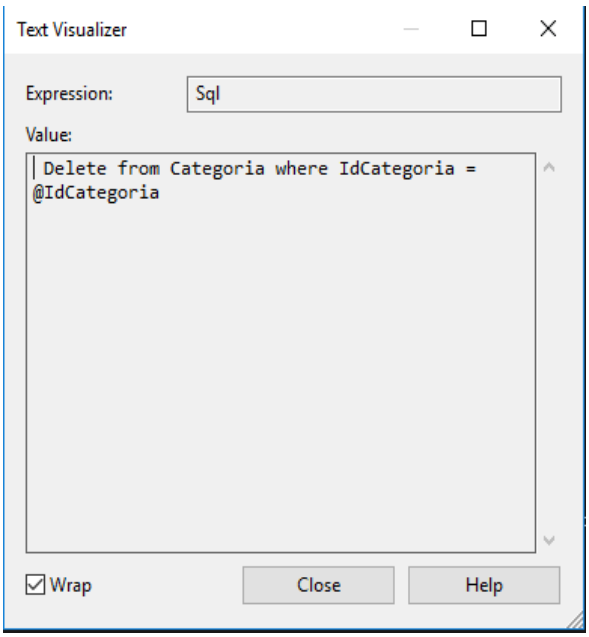

Figura 31 - SQL para deletar dados

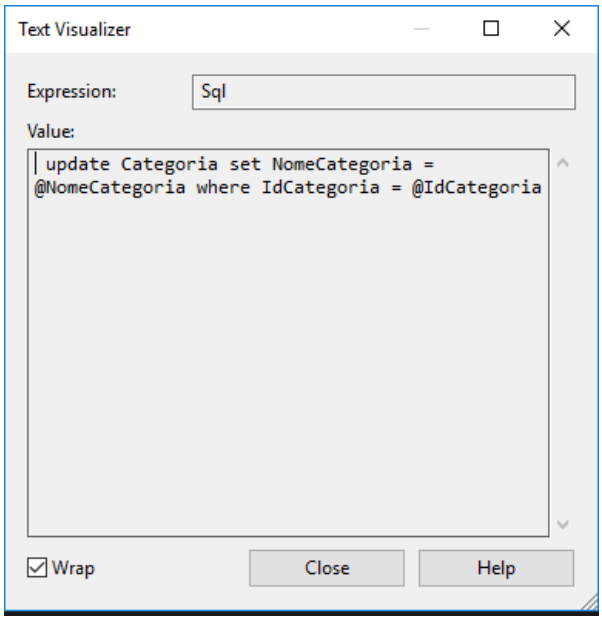

Figura 32 - SQL para atualizar os dados

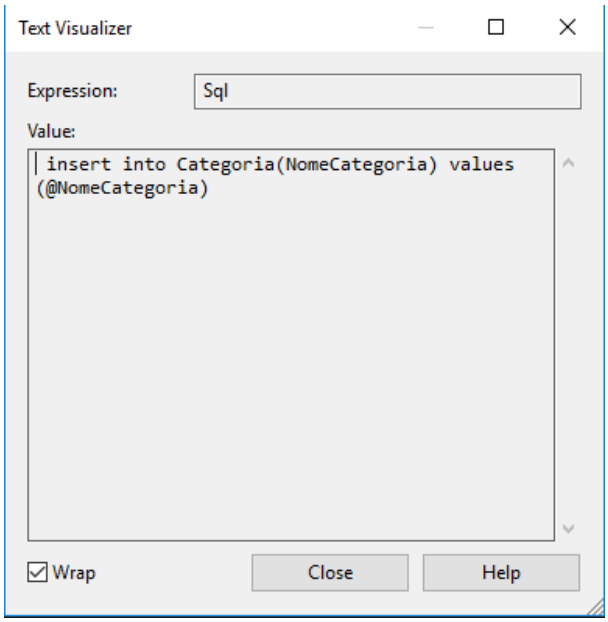

Figura 33 - SQL para salvar os dados

Como apresentado nas figuras anteriores, as SQLs geradas pelo *framework* são simples. Nota-se que as SQLs geradas pelo *framework* quase não têm diferença de SQLs gerados manualmente pelo programador, como é apresentado na Figura 34, a seguir.

```
insert into categoria(nomecategoria)
values (@parm_1)

update categoria set nomecategoria = @parm_1
where idcargoria = @parm_2

delete from categoria
where idcategoria = @parm_1
```

Figura 34 - SQLs gerados manualmente por um programador

Nota-se uma pequena diferença entre os SQLs, pois nesse caso o programador utilizou o padrão de @param e número para identificar os valores, já no caso *framework* é utilizado o nome do atributo junto com @.

As SQLs geradas obedecem ao padrão de cada banco de dados, pois por mais que na maioria das vezes eles são iguais, há algumas diferenças particulares de cada Banco de dados, como por exemplo da paginação. No Oracle não existe as *tag offset* e *rows fetche next* que são usados no Microsoft SQLServer para a realização da paginação, No Oracle utiliza-se a *tag row num*. Na mesma situação o MySQL e PostgreSQL utilizam as *tag offset* e *limite* para realização da paginação.

4.5 Comparando *Framework* com *Entity Framework*

Para verificar a eficiência do *framework* desenvolvido o mesmo será comparado com um *framework* presente no mercado, o *Entity Framework*. Para realizar a comparação, foi construído um protótipo usando o *Entity Framework* com as mesmas características do protótipo apresentado anteriormente, construído usando o *framework* proposto. Sendo assim, esse protótipo tem as mesmas tabelas e estrutura do projeto, mudando apenas as peculiaridades de cada *framework* como mapeamento dos objetos e conexão com banco de dados.

4.5.1 Mapeamento no *Entity Framework*

No *Entity Framework* é possível mapear o banco de dados de duas maneiras, utilizando *data annotation* e *fluente api*. No caso do *fluente api* utiliza-se os métodos encadeados fora das entidades para realizar o mapeamento do banco de dados, como apresentado na Figura 35.

```
1  HasKey(x => x.ClienteId);
2  Property(x => x.Nome)
3       .HasMaxLength(150)
4       .HasRequired();
```

Figura 35 – Exemplo de mapeamento do banco de dados com *fluent api*

Contudo, no protótipo foi utilizado as *data annotation* para realizar o mapeamento do banco de dados. O conceito da utilização das *data annotation* no mapeamento com *Entity Framework* funciona como no *framework* proposto nesse trabalho, com a diferença de que

quando utilizando SQLServer, o *Entity Framework* não tem necessidade da *data annotation Column*. A seguir, na Figura 36, é apresentado o mapeamento da classe Categoria.

```
[Table("Categoria")]
24 references
public class Categoria
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    8 references | 0 exceptions
    public int IdCategoria { get; set; }
    5 references | 0 exceptions
    public string NomeCategoria { get; set; }
}
```

Figura 36 - Mapeamento da tabela Categoria com *Entity Framework*

Contudo essa não é única diferença quando é preciso mapear classes com *Foreing Key*. É necessário utilizar o comando virtual do C# que informa que objeto pode ser sobrescrito. Já no *framework* desenvolvido nesse trabalho não tem a necessidade da utilização do comando virtual, e nem um atributo para representar da *Foreing Key*, pois o *framework* busca o valor direto da classe referenciada, essas diferenças são apresentadas nas Figura 37 e Figura 38.

```
[Table("Produto")]
22 references
public class Produto
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    6 references | 0 exceptions
    public int IdProduto { get; set; }
    2 references | 0 exceptions
    public int IdTipoProduto { get; set; }
    5 references | 0 exceptions
    public decimal Preco { get; set; }
    5 references | 0 exceptions
    public string NomeProduto { get; set; }
    5 references | 0 exceptions
    public string Descricao { get; set; }

    [ForeignKey("IdTipoProduto")]
    3 references | 0 exceptions
    public virtual TipoProduto TipoProduto { get; set; }
}
```

Figura 37 - Mapeamento da tabela Produto com *Entity Framework*

```
[Table("Produto")]
9 references
public class Produto
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    [Column("IdProduto")]
    2 references | 0 exceptions
    public int IdProduto { get; set; }
    [Column("Preco")]
    0 references | 0 exceptions
    public decimal Preco { get; set; }
    [Column("NomeProduto")]
    0 references | 0 exceptions
    public string NomeProduto { get; set; }
    [Column("Descricao")]
    0 references | 0 exceptions
    public string Descricao { get; set; }
    [ForeignKey("IdTipoProduto")]
    0 references | 0 exceptions
    public  TipoProduto TipoProduto { get; set; }
}
```

Figura 38 - Mapeamento da tabela Produto com *framework* proposto

Como apresentado anteriormente, na Figura 37 o *Entity Framework* precisa informar um atributo para representar a *Foreing Key*. Na classe Produto, já mapeada, a mesma possui um atributo IdTipoProduto e outro atributo virtual chamado TipoProduto. Já utilizando *framework* apresentado nesse trabalho é utilizado apenas atributo TipoProduto, pois o framework identifica o atributo pela *data annotation Foreign Key*.

Em tabelas com entidade fraca, o *Entity Framework* tem a necessidade de passar a ordem das colunas do banco para informar as chaves primárias. Já no *framework* desenvolvido é necessário apenas informar quais atributos são chaves primárias utilizando a *data annotation* Key. A seguir, na Figura 39, é apresentado o mapeamento da tabela CategoriaProduto com *Enity Framework*, que é uma entidade fraca no banco de dados. Na Figura 40 segue o mesmo mapeamento da tabela CategoriaProduto utilizando o *framework* proposto.

```
[Table("CategoriaProduto")]
18 references
public class CategoriaProduto
{
    [Key, Column(Order = 0)]
    5 references | 0 exceptions
    public int IdCategoria { get; set; }
    [Key, Column(Order = 1)]
    2 references | 0 exceptions
    public int IdProduto { get; set; }

    [ForeignKey("IdProduto")]
    1 reference | 0 exceptions
    public virtual Produto Produto { get; set; }
    [ForeignKey("IdCategoria")]
    1 reference | 0 exceptions
    public virtual Categoria Categoria { get; set; }
}
```

Figura 39 - Mapeamento da tabela CategoriaProduto com *Entity Framework*

```
[Table("CategoriaProduto")]
8 references
public class CategoriaProduto
{
    [Key]
    [ForeignKey("IdProduto")]
    3 references | 0 exceptions
    public  Produto Produto { get; set; }
    [Key]
    [ForeignKey("IdCategoria")]
    6 references | 0 exceptions
    public  Categoria Categoria { get; set; }
}
```

Figura 40 - Mapeamento da tabela CategoriaProduto com *framework* proposto

4.5.2 Repositório *Entity Framework*

No *Entity Framework* pode-se ou não criar um repositório genérico. Esse repositório é uma classe onde todos os métodos padrões como salvar, deletar, atualizar e consultar estariam. Esse repositório é criado para que não seja necessário criar todos esses métodos em todas as classes que comunicam com banco de dados. No *framework* desenvolvido esse repositório já é disponibilizado para usuário. Para o protótipo do *Entity Framework*, foi criado uma classe para esse repositório para comparar os métodos básicos entre o *framework* proposto e *Entity Framework*. Os métodos presentes no repositório são: salvar, atualizar, consultar por id, consultar todos e método para deletar.

Para operações como salvar, deletar e atualizar no *Entity Framework* é necessário usar o comando SaveChanges. Sem esse comando as operações não são salvas no banco de dados. A seguir, na Figura 41, é apresentado o método salvar utilizando o *Entity Framework*.

```
4 references | 0 exceptions
public virtual void Salvar(List<T> list)
{
    try
    {
        list.ForEach(m => ctx.Set<T>().Add(m));
        ctx.SaveChanges();
    }
    catch (Exception ex)
    {
        throw new Exception("Erro ao salvar dados: " + ex.Message);
    }
}
```

Figura 41 - Método salvar *Entity Framework*

No *Entity Framework* é possível passar expressões *lambda* para realizar algumas operações como filtro em uma consulta. Expressões *lambda* segundo a Microsoft (2016) "É uma função anônima que você pode usar para criar *delegates* ou árvores de expressões. Usando expressões *lambda*, você pode escrever funções locais que podem ser passadas como argumentos ou retornadas como o valor de chamadas de funções. "

Na Figura 42 é apresentado o método de ConsultarTodos. Esse método recebe como parâmetro um *predicate*, que é uma expressão *lambda* e esse *predicate* contém as condições no qual o *Entity Framework* irá utilizar para montar o filtro na consulta no banco de dados.

```
4 references | 0 exceptions
public virtual List<T> ConsultarTodos(Func<T, bool> predicate, int? offset, int? limite)
{
    try
    {
        if (offset != null && limite != null)
            return ctx.Set<T>().Where(predicate).Skip(Convert.ToInt32(offset)).Take(Convert.ToInt32(limite)).ToList();
        return ctx.Set<T>().Where(predicate).ToList();
    }
    catch (Exception ex)
    {
        throw new Exception("Erro ao consultar todos: " + ex.Message);
    }
}
```

Figura 42 - Método consultar todos *Entity Framework*

4.5.3 Conexão *Entity Framework*

Para utilizar o *Entity Framework* é necessário criar uma classe para conexão com banco de dados. Essa classe deve herdar da classe DbContext, que é uma classe do *Entity Framework*, onde estão todas as funções padrões da conexão, como controladores de acesso à base de dados e funções como *insert*, *delete* entre outros. No construtor da classe de conexão deve-se informar uma *String* de conexão para que *Entity Framework* possa criar o mapeamento da base de dados.

Na classe de conexão é necessário instanciar os *DbSet* das classes mapeadas. Eles são necessários pois o *Entity Framework* os utiliza para fazer o vínculo entre as classes mapeadas e banco de dados. Na Figura 43 é apresentada a classe de conexão criada para o protótipo do *Entity Framework*.

```
3 references
public class Conexao : DbContext
{
    1 reference | 0 exceptions
    public Conexao() : base("strHomologacao")
    {

    }

    0 references | 0 exceptions
    public DbSet<Produto> Produto { get; set; }
    0 references | 0 exceptions
    public DbSet<TipoProduto> TipoProduto { get; set; }
    0 references | 0 exceptions
    public DbSet<Categoria> Categoria { get; set; }
    0 references | 0 exceptions
    public DbSet<CategoriaProduto> Evento { get; set; }


}
```

Figura 43 - Classe de conexão do protótipo do *Entity Framework*

Na classe de conexão criada para o *Entity Framework*, é possível criar algumas configurações para informar que *Entity Framework* será responsável pela geração do banco de dados a partir das classes mapeadas.

4.5.4 SQLs gerados pelo *Entity Framework*

Em todos os SQLs de consulta gerados pelo *Entity Framework* foi acrescentado uma palavra chamada *extent* e uma numeração. Essa palavra é gerada para identificar as tabelas quando é montado o SQL. Para realizar a comparação dos SQLs gerados pelo *Entity Framework* e o *framework* desenvolvido, foi gerado SQL para inserção de registro na tabela Categoria que é apresentado na Figura 44.

```
INSERT [dbo].[Categoria]([NomeCategoria])
VALUES (@0)
```

Figura 44 - SQL de inserção na tabela categoria com *Entity framework*

Também foi gerado um SQL para consulta na tabela Categoria utilizando o *Entity Framework*. Esse SQL é apresentado na Figura 45.

```
{SELECT
    [Extent1].[IdCategoria] AS [IdCategoria],
    [Extent1].[NomeCategoria] AS [NomeCategoria]
    FROM [dbo].[Categoria] AS [Extent1]}
```

Figura 45 - SQL de consultar todos na tabela categoria com *Entity Framework*

O SQL de inserção gerado pelo *Entity Framework*, e apresentado na Figura 44, não é muito diferente do SQL gerado pelo *framework* proposto, apresentado na Figura 33. Contudo, o SQL para consulta de todos os registros da tabela apresenta algumas diferenças, além da palavra *extent*. No SQL gerado pelo *Entity Framework* foi necessário a utilização da operação AS, que no script SQL serve para renomear os atributos retornados pela consulta. Isso torna o SQL maior se comparado ao SQL gerado pelo *framework* proposto. Essa diferença pode ser

notada comparando as Figura 45, que apresenta SQL gerado pelo *Entity Framework* e a Figura 30 que apresenta o SQL gerado pelo *framework* proposto.

4.5.5 Vantagens do *Framework* proposto em relação *Entity Framework*

O *framework* desenvolvido apresenta algumas vantagens em relação ao E*ntity Framework*. Essas, vantagens são: os SQLs gerados para consulta são mais simples, o repositório com os métodos padrões já está pronto sendo necessário apenas a herança do mesmo; o *framework* não tem necessidade de ter os *DbSets* na conexão para associar as classes mapeadas com o banco de dados; e o *framework* apresenta de maneira simples a possibilidade de atualização de vários bancos de dados em um mesmo projeto, sem a necessidade de criar outras classes para conexão.

4.5.6 Desvantagens do *Framework* proposto em relação *Entity Framework*

O *framework* proposto também apresenta algumas desvantagens em relação ao *Entity Framework*, elas são: não é possível utilizar expressões *lambdas* para a realização de consultas no banco de dados; não é possível criar banco de dados pelo código, como pode ser realizado no *Entity Framework;* e no *framework* proposto só é possível mapear o banco de dados com *data annotation.* Já no *Entity Framework*, como já foi exposto anteriormente, o mesmo possui duas maneiras de mapear o banco de dados: *data annatation* e *fluent api*.

4.6 Pontos positivos do *Framework* proposto

Na utilização do *framework* proposto pode-se observar diversos pontos positivos, como padronização de código e agilidade no desenvolvimento. Contudo esses pontos estão presentes na maioria dos *frameworks*. No *framework* apresentado existem diversos outros pontos positivos como: a facilidade do mapeamento da base de dados, a possibilidade de utilizar várias conexões com banco de dados diferentes, diversos repositórios para utilização em classes com responsabilidades diferentes e SQLs gerados são mais simples se comparados com outros *frameworks*.

4.7 Pontos Negativos do *Framework* proposto

Na utilização do *framework* proposto, como em vários outros *frameworks*, pode-se verificar alguns pontos negativos como a dependência dos métodos que já estão prontos, além da dificuldade de implementar métodos que fogem as regras do *framework*. No *framework* proposto pode-se verificar a dependência de *data annotation* para utilização dos recursos do mesmo. Também, como ponto negativo do *framework* proposto, deve-se criar o banco de dados antes de iniciar a utilização do mesmo, pois o mesmo não tem possibilidade de criar a base de dados a partir do mapeamento das classes como em outros *frameworks* no mercado.

4.8 Resultados Obtidos

No desenvolvimento do *framework* foram obtidos os resultados conforme exposto no objetivo desse trabalho. O mesmo apresenta um código simples e de fácil entendimento e as SQLs geradas pelo *framework* são e simples.

O *framework* proposto não é dependente de *providers* para realizar conexões com bancos de dados, como em outros *frameworks.* Ele depende exclusivamente apenas das DLLs padrões de conexão de cada banco de dados.

O *framework* apresenta uma estrutura na qual ele consegue atender diversos projetos diferentes de forma simples. Tal estrutura também permite extensões e novas implementações aumentando assim sua utilidade.

O framework também possibilita utilização, de forma simples, várias conexões em bancos de dados diferentes. Esse é um recurso muito importante para projetos que utilizam a arquitetura de microserviços.

5 CONCLUSÃO

O presente trabalho propôs demostrar a criação e implementação de um *framework* para auxiliar no desenvolvimento de *software* utilizando a linguagem de programação C#. Este *framework* facilita o desenvolvimento por meio de repositórios que podem ser herdados. Esses repositórios são:

- RepositorioControllerWeb: Responsável por fazer comunicação entre as *views* e os *controllers*.
- ReposiotiroBLL: Responsável pela regra de negócio do sistema.
- RepositorioDAL: Responsável pela comunicação e interação com os bancos de dados.

Para utilizar esses repositórios é necessário que projeto esteja estruturado em camadas. Essas camadas devem estar dividas em camada de dados, camada de negócio e camada de apresentação na qual aplicação web estará rodando.

Também é necessário que objetos estejam mapeados de forma adequada para que *framework* possa ler seus meta-dados e assim realizar as operações necessárias com esses meta-dados, como a geração de SQLs dinâmicos.

Para o desenvolvimento do *framework* proposto nesse trabalho foram utilizadas as seguintes tecnologias: linguagem de programação C#, IDE de desenvolvimento Visual Studio, conceitos de UML e Arquitetura de Camadas.

Esse trabalho foi resultado da necessidade de criação de *framework* que possibilitasse de forma simples a interação com vários bancos de dados sem a necessidade de utilização de *providers*, além de minimizar o trabalho do desenvolvedor com questões operações simples em banco de dados, como cadastros, atualizações e remoções.

Para trabalhos futuros o *framework* pode implementar funções apresentadas em outros *frameworks*, como criação do banco de dados pelas classes mapeadas e utilizar expressões *lambda* para gerar SQL com filtros, apresentado no *Entity Framework*.

6 REFERÊNCIAS

ABDALA, Daniel Duarte. WANGENHEIM, Aldo Von. Conhecendo o Smalltalk. VISUAL BOOKS, 2002. 284p.

ABREU, Luis, Asp.Net 4.5.1 - One Asp.Net, Owin, Identity, Mvc, Web Api e Signalr. Fca Editora. 2014. 236p.

AVILLANO, Israel De Campos. Object Pascal para Delphi. Ciência Moderna. 2009. 192p.

CARNEIRO, Cristiane. Frameworks de aplicações orientadas a objetos – uma abordagem iterativa e incremental. 2003. 121 f. Dissertação (Mestrado em Redes de Computadores) – Universidade Salvador, Salvador, 2003.

COELHO, Pedro. Programação em Java. Fca, 2014. 528p.

CORDOSO, gabriel shade, Microsoft Kinect: Crie aplicações interativas. 2013. 181.

DAHL, Ole-Johan. NYGAARD, Kristen. SIMULA- An Algol Based Simulation Language." Comm. ACM, 9 (9), p. 671-678, 1966.

DEITEL, Harvey. C++ Como Programar. Pearson. 2006. 1216p.

FURGERI, Sérgio. Programação Orientada a Objetos: conceitos e técnicas. São Paulo: Érica ,2015.

GUEDES, Gilleanes. UML 2: Uma abordagem pratica. São Paulo: Novatec. 2009. 484p.

JOHNSON, Ralph. E.; FOOTE, B. Designing Reusable Classes. Journal of Object-Oriented Programming, 1988. Disponível em: <https://www.cse.msu.edu/~cse870/Input/SS2002 /MiniProject/Sources/DRC.pdf> Acesso em: 30 de maio 2001.

JORGE, Marcos. Delphi 7. Passo a Passo. Pearson, 2013. 176p.

LECHETE, Ricardo Rodrigues. Desenvolvendo para iPhone e iPad. São Paulo: Novatec. 2016. 640p.

LECHETE, Ricardo Rodrigues. Google Android: Aprenda a Criar Aplicações para Dispositivos Móveis com o Android SDK. São Paulo: Novatec. 2013. 824p.

LEITE, Alessandro. Frameworks e Padrõ**es** de Projeto. Disponível em: <http://www.devmedia.com.br/frameworks-e-padroes-de-projeto/1111>. Acesso em: 05 de janeiro de 2016.

MAGALHÃES, Alberto. SQL Server 2014. Curso Completo. FCA, 2015. 624p.

MANZANO, José Augusto N. G. Estudo Dirigido de Microsoft Visual Basic Community 2015. Érica. 2015. 160p.

MENEZES, Nilo Ney Coutinho. Introdução à Programação com Python. São Paulo: Novatec. 2014. 328p.

MICROSOFT, Expressões lambda (Guia de Programação em C#) .2016. Disponível em: <https://msdn.microsoft.com/pt-br/library/bb397687.aspx>. Acesso em 03/09/2016.

MICROSOFT, SharePoint Deployment guide for Microsoft SharePoint 2013. 2014. 1156p.

MICROSOFT, Visão geral e breve análise do ADO.NET Entity Framework. 2015. Disponível em: <https://msdn.microsoft.com/pt-br/data/aa937709>. Acesso em 06/06/2016.

MICROSOFT, Visual Studio IDE. 2016. Disponível em: <https://msdn.microsoft.com/pt-BR/library/dn762121.aspx>. Acesso em 21/01/2016.

MICROSOFT. C#. 2016. Disponível em: <https://msdn.microsoft.com/pt-br/library/kx37x362.aspx>. Acesso em 16/10/20116.

MILANI, André. MySQL: Guia do Programador. São Paulo: Novatec. 2007. 400p.

MILANI, André. POSTGRESQL: Guia do Programador. São Paulo: Novatec. 2008. 392p.

MORRISON, Michael. Use a Cabeça! Javascript. Alta Books. 2008. 640p.

PEREIRA, Michael Henrique R. AngularJS. Uma Abordagem Prática e Objetiva. São Paulo: Novatec. 2014. 208p.

PORTAL EDUCANDO. História e características da linguagem C#. 2008. Disponível em: <http://www.portaleducacao.com.br/informatica/artigos/6137/historia-e-caracteristicas-da-linguagem-c>. Acesso em: 19 de outubro de 2015.

SANT'ANNA, Mauro. C#, uma linguagem para o novo milênio. 2000. Disponível em: < https://msdn.microsoft.com/pt-br/library/cc518016.aspx>. Acesso em: 20 de outubro de 2015.

SILVA, Mauricio Samy, JQuery. A Biblioteca do Programador Javascript. São Paulo: Novatec. 2013. 544p.

SURHONE, Lambert M. Turbo Pascal. BETASCRIPT PUB, 2010. 104p.

TULLOCH,mitch. Introducing Windows Azure: for it professionals.2013. 142p.

WATSON, John. OCA Oracle Database 11g Administração I. Guia do Exame 1Z0-052. Bookman, 2009. 704p.

WOLFF, Eberhard. Microservices: Flexible software architectures. Createspace pub. 2016.